Assignment 8

Due May 14, 5:00 p.m.

For this final assignment, you will call upon the ideas you’ve learned – including abstraction, nondeterminism, recursion, iteration, destructive programming, structures, and vectors – for an urgent task. Robot loves Kitten, but Kitten is missing, lost in a world filled with useless, non-Kitten objects. You must help Robot find Kitten.

This quest to make everything right again is a game – or “Zen simulation” – first written in 1997 by Leonard Richardson. Since then, it’s been ported to many computer systems and implemented in many programming languages. You can even play it in your web browser ([http://robotfindskitten.org/play/robotfindskitten](http://robotfindskitten.org/play/robotfindskitten)). But we know that the best programs are written in Scheme, so we’re going to implement our own version of robotfindskitten!

To make this easier, you are provided with a game template, `asmt08.scm` and instructions, below, on how to complete the game. While the full working game will be quite short – only be a few hundred lines of code! – the parts need to be complete in the described order, and making everything work together may require some time spent debugging. If the instructions are unclear or you get stuck, ask for help!

Part 1

If you wish to find Kitten from scratch, first you must create the universe. At the bottom of the template file is a call to the `big-bang` procedure, which does just that. This procedure is provided by the `2htdp/universe` library, and it handles the basic game mechanism: a loop that continues until the end-of-game condition is reached, calling procedures to draw the screen and handle the player’s inputs.

You can see that `big-bang` takes as its inputs `*world*`, which will be our representation of the game world, and the names of the functions `draw-world` and `handle-input`. In addition, it takes an anonymous function that tells it to stop when the global variable `|game-over|` is true, and an argument specifying that it should close the graphics window when the game stops.

At the moment, all that happens is that `draw-world` makes a black window. It can’t do anything more because `*world*` is just a void. Let’s fix that.

---

1 Graphics? While robotfindskitten uses the finest and simplest ASCII art the ’90s had to offer, it can’t be reproduced in the Interactions Window of DrRacket. Instead, we use a separate window for drawing graphics, as we did to make our impressive arches and pyramids. However, for this assignment, we are using a richer graphics library, `2htdp/image`. 
a. Define a destructive function `make-world!` that takes no arguments but modifies the global variable `*world*`. We need `*world*` to represent what’s at each position in the two-dimensional world. To do this, we can divide the window into a grid of “cells”, each of which can (1) be empty, (2) be occupied by Robot, or (3) be occupied by an object, which may or may not be Kitten. The number of rows and the number of columns should both be `*cells-per-side*`. Since the size of the world won’t change, but its contents will, this makes a vector the ideal choice. The `make-world!` procedure should set `*world*` to be a vector of the right size to hold these cells.

Once you’ve defined `make-world!`, uncomment the call to this procedure that’s above the call to `big-bang`. Now when the game starts, the world will be vector rather than a void.

b. To make sure our world is being correctly divided into cells, let’s draw their edges. The `draw-world` procedure generates `scene`, which is just a black space of the right size, and returns it.

Update `draw-world` to use iteration to go through each cell and draw it. To do this, you need the `(x, y)` pair indicating the cell’s position in the world. To do this, you can use nested iteration, as we did in class to draw a checkerboard.

To draw a cell, you can update the value of `scene` to be the result of calling the provided procedure `(draw-cell scene x y)`. When this works, you’ll have a grid of lines giving 20 squares per side.

If you want to change the colors of the background or the lines, change the strings “black” and “slategray” to any of the options listed at https://docs.racket-lang.org/draw/color-database__.html

Part 2

The world we’ve created is pretty lonely. Next we’ll populate it with objects for Robot to explore.

a. The objects in the world will need various properties: the symbol used to display it, the color of the symbol, a message to print when Robot bumps into it, and a Boolean value for whether or not it is Kitten.

Define a structure, `object` with these fields allowing you to make objects, e.g,

> `(make-object "%" "red" "A pile of broken parts" #f)`

#<object>

b. To make the game interesting, Robot should have no idea what an object is before bumping into it. Therefore, objects should have a randomly selected symbol, color, and message. The template file includes vectors for each of these, which you’re encouraged to modify.

---

2 Note: This is the position of the cell in the visual grid of cells, not the actual pixel locations.

3 Having an amusing collection of objects is most of the fun of robotfindskitten. You can see the original collection of messages here: https://github.com/leonardr/robotfindskitten-dos/blob/master/rfk_1600000.160-robot.cpp
Define a procedure `random-object` that takes no inputs and returns an object with a randomly chosen symbol, color, and message. No object generated by `random-object` should (initially) be Kitten.

**Note:** Since this requires selecting a random element from each of these vectors, you should write a helper function, e.g., `random-element`, to do this rather than repeat the code.

c. We want to find random empty cells to put objects into. Define a procedure `random-empty-cell` that takes no inputs and returns a the index of an empty cell as a list `(x y)`. If the randomly selected cell is already set to an object, `random-empty-cell` should try again by calling itself recursively.

To convert the `x` and `y` coordinates into an index in the `*world*` vector, you can use the provided `world-index` procedure, e.g.,

```
> (world-index 3 6)
123
```

d. Update your `make-world!` procedure to use iteration to set random empty cells of the `*world*` vector to new random objects. It should do this `*num-objects*` times.

e. While we've populated our world vector, we can't yet see those objects. Update the `draw-world` procedure so that it calls the provided `draw-object` procedure on each cell of `*world*`. This procedure takes as input the current scene, the value of the cell (which may or may not be an object), and the `x` and `y` coordinates of the cell. It returns a scene, which should be used to overwrite the existing scene, as you did with `draw-cell`.

At this point, your grid should include random symbols of different colors in various cells, and it should change each time you run your code.

**Part 3**

We now have a world filled with non-Kitten objects. We need to add Kitten and Robot.

a. Kitten is represented with an object, but has the `kitten?` field in the structure set to `#t`. We want to add exactly one Kitten each time the world is generated. The easiest way to do this is to check if the iteration index being used to generate all of the `*num-objects*` objects is 0. If so, the object being generated should be (destructively) set to be the Kitten, and the name/message for it should be set appropriately before it is added to the `*world*` vector.

b. Robot should not be stored inside the `*world*` vector or else we would need to search through the vector to move. Create a global variable `*robot*` which is a list of the `x` and `y` coordinates where the Robot currently is. The initial value should be the result of calling `random-empty-cell`.

c. To show where Robot is, update `draw-world`: Before returning the scene, set it to be the result of calling `(draw-robot scene)`. 
Part 4

The world has Kitten, Robot, and a variety of randomly generated and placed non-Kitten objects. In this part, we will make it possible for Robot to move through the world, searching for Kitten.

a. The handle-input procedure decides what to do with the player's keyboard inputs. By default it lets you type “q” to quit. There are additional cases for moving the robot that are commented out. Uncomment these!

b. Define the move-robot! procedure to update the robot's position. If the input is 'left, the robot's x position should be updated to x - 1, and if it's 'right, x should be updated to x + 1. If it's 'up, y should be changed to y - 1, and if it's 'down, y should be changed to y + 1. Check that with this change you can now move the robot.

c. Make your move-robot! procedure detect when the robot would be moving out of the world. In these cases, do not update the robot's position. Instead, print a message to the Interactions Window (e.g., "This is not an exit").

d. Update move-robot! to detect whether there is an object already in the cell of *world* that it would be moving into. You may wish to define a helper function, cell-empty? to do this that takes inputs x and y and returns #t or #f. If the cell is not empty, the robot should not move into it. Instead, the message stored for that object should be printed.

e. Lastly, we need to recognize when the object that Robot has encountered is Kitten. You may wish to move the functionality of printing messages and recognizing Kitten into a function bump-into!. If the object is Kitten (i.e., has the kitten? field set to #t), in addition to printing the message, which should state that Robot found Kitten, *game-over* should be set to #t.

Part 5

Find Kitten!

Submitting

Don't forget to submit your work using the submit101 command!

submit101 g-asmt08 asmt08

(If the name of your directory is different from asmt08, change asmt08 to whatever the name of your directory is.)

For this assignment you don't need to submit an interactions file.